CEWES MSRC/PET TR/99-26

Reading Sequential Unformatted CRAY C90 Files on an SGI Origin

by

James B. White, III

05h00299

Work funded by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through

Programming Environment and Training (PET)

Supported by Contract Number: DAHC94-96-C0002
Nichols Research Corporation

Reading Sequential Unformatted CRAY C90 Files on an SGI Origin

James B. White III

May 3, 1999

1 Introduction

The CRAY C90 and SGI Origin have different formats for the binary representation of Fortran data types, and they have different layouts for binary data in files. Therefore, a Fortran program compiled on an Origin cannot read a sequential unformatted file generated on a C90 without translating from the C90 file structure and data-type format. This document describes how to perform the required translation assuming only an Origin system is available.

The issues surrounding the translation of C90 sequential unformatted files fall into three categories: the `assign` command, the default conversion, and the `cry2mips` Fortran library function. A example module and test programs using `cry2mips` appear in Section 5.

2 The `assign` Command

The `assign` command modifies the attributes assigned to a file, Fortran I/O unit, file type, or any file whose name matches a specified pattern. The modified attributes cause the Fortran I/O intrinsics, `read` and `write`, to treat the specified files differently.

Programs compiled with the MIPSpro Fortran 90 compiler (`f90`) automatically change their I/O behavior at runtime based on the active `assign` modifications. Programs compiled with the MIPSpro F77 compiler, however, will only respond correctly to `assign` modifications if they are compiled with the "`-craylibs`" option.

The `assign` command saves information on modified attributes in a single file for each user. On the C90, the location of this file is set by default to be "`$TMPDIR/.assign`". On the Origin, however, the location of this file must be set by the user through the `FILENV` environment variable. The following examples cause `assign` attributes to reside in the file "`.assign`" within the user's home directory, for `ksh` and `csh`, respectively.

```
export FILENV=$HOME/.assign
setenv FILENV $HOME/.assign
```

With `FILENV` set, the `assign` command can specify that a particular file or Fortran unit is in the C90 format. Full specification of a C90 file or unit requires two options to `assign`. The first option, "`-F cos`", indicates the "COS blocked structure", the default file structured used by sequential unformatted files on the C90. The second option, "`-N cray`", indicates that data are in the Cray Research format, not the IEEE Standard format. The following example sets the appropriate options for a given file.

```
assign -F cos -N cray filename
```

Once this `assign` statement is executed, any Cray file with the specified name can be read by a Fortran program executed on the same Origin by that user. This also means Origin files with the specified name *cannot* be read until the `assign` modifications are removed (using "`assign -R`"). The `assign` command sets runtime options for the Fortran I/O library, so a program does not need to be modified or recompiled to use the information provided by `assign`.

3 Default Conversion

Conversion of `character` and `integer` variables between Cray files and Origin programs is straightforward and automatic with "`assign -F cos`". Both 32-bit and 64-bit `integer` variables on the Origin are written as 64-bit values in a Cray file. Similarly, `integer` values in a Cray file, which are always 64-bit, can be read into 32-bit or 64-bit variables. Of course, reading a 64-bit value into a 32-bit variable gives an incorrect result if the value is too large (positive or negative) to fit in 32 bits.

The "`-N cray`" argument to the `assign` command specifies that Fortran `read`s and `write`s should translate between the default size and format of Fortran floating-point types for the C90 and Origin. Floating-point values on the Origin are stored in IEEE format, while floating-point values on the C90 are stored in a unique Cray format. Under the "`-N cray`" translation, 32-bit floating-point variables are written as 64-bit Cray values, and 64-bit variables are written as 128-bit "`double precision`" Cray values.

The automatic promotions for writing to a Cray file mirror automatic demotions forced on reading from a Cray file. A `read` of a 64-bit Cray floating-point value must use a 32-bit variable, not a 64-bit variable. A `read` of a 128-bit Cray value can use a 32-bit or 64-bit variable, but not a 128-bit `real*16` variable. Therefore, the automatic numeric conversion by `assign` forces a loss of precision that cannot be avoided using only `read` statements. This loss of precision is likely to be unacceptable for many applictions. Values and variables of the type `complex` have similar unacceptable constraints on reading and writing.

4 The `cry2mips` Fortran Library Function

As of this writing, the author knows of no way to read a 64-bit Cray value directly into a 64-bit IEEE variable. The Fortran library function `cry2mips` provides an indirect way of doing this, however. The first step is to read the 64-bit Cray floating-point value into a 64-bit `integer`. The `cry2mips` function can then convert the Cray value stored in the `integer` to a 64-bit IEEE value stored in an appropriate `real` variable.

Using two 64-bit `integers`, `cry2mips` can make the equivalent `complex` conversion. Using an array of 64-bit `integers`, `cry2mips` can convert arrays of variables. The following code fragments illustrate typical conversions. For details on the arguments to `cry2mips`, see "`man cry2mips`". A reverse translation function, `mips2cry`, is also available.

One 64-bit `real` value:

```
 integer status
 integer, parameter :: real_type = 3
 integer, parameter :: real_size = 64
 integer(8) ix
 real(8) x
 open(unit=33, file="cray.dat", form="unformatted")
 read(33) ix
 status = cry2mips(real_type, 1, ix, 0, x, 1,
$   real_size, real_size)
 if (status .ne. 0) stop "CRY2MIPS ERROR"
```

An array of 64-bit `real` values:

```
 integer status
 integer, parameter :: real_type = 3
 integer, parameter :: real_size = 64
 integer, parameter :: n = 1000
 integer(8) ix(n)
 real(8) x(n)
```

```
 open(unit=33, file="cray.dat", form="unformatted")
 read(33) ix(:)
 status = cry2mips(real_type, n, ix(1), 0, x(1), 1,
$   real_size, real_size)
 if (status .ne. 0) stop "CRY2MIPS ERROR"
```

One 128-bit `complex` value:

```
 integer status
 integer, parameter :: complex_type = 4
 integer, parameter :: complex_size = 128
 integer(8) iz(2)
 complex(8) z
 open(unit=33, file="cray.dat", form="unformatted")
 read(33) iz(:)
 status = cry2mips(complex_type, 1, iz(1), 0, z, 1,
$   complex_size, complex_size)
 if (status .ne. 0) stop "CRY2MIPS ERROR"
```

An array of 128-bit `complex` values:

```
 integer status
 integer, parameter :: complex_type = 4
 integer, parameter :: complex_size = 128
 integer, parameter :: n = 1000
 integer(8) iz(2,n)
 complex(8) z(n)
 open(unit=33, file="cray.dat", form="unformatted")
 read(33) iz(:,:)
 status = cry2mips(complex_type, n, iz(1,1), 0, z(1), 1,
$   complex_size, complex_size)
 if (status .ne. 0) stop "CRY2MIPS ERROR"
```

5 Example

A program for generating a test file on a C90 appears in Section 5.1. It generates the file "`cray.dat`", which contains `characters`, `reals`, `complexs`, and an `integer`.

Section 5.2 shows a module, `cray_binary`, that provides subroutines for reading `real` and `complex` variables and arrays. The program in Section 5.3 `read_cray_binary`, uses this module to read the file "`cray.dat`" when it has been moved to an Origin.

Assuming `FILENV` is set, the following `assign` command makes "`cray.dat`" readable by `read_cray_binary`.

```
assign -F cos -N cray cray.dat
```

The output of `read_cray_binary` should be similar to the following.

```
Cray Data:
A = 3.1415926535897967
C = (3.1415926535897967,3.1415926535897967)
N = 10
X = 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.,  10.
Z = (1.,-1.),  (2.,-2.),  (3.,-3.),  (4.,-4.),  (5.,-5.),
(8.,-8.),  (9.,-9.),  (10.,-10.)
```

5.1 Program write_cray_binary

The following Fortran program, write_cray_binary, can run on a CRAY C90. It generates an unformatted file called "cray.dat".

```
program write_cray_binary

implicit none

integer, parameter :: n = 10
character(len=10) title
integer i
real a, x(n)
complex c, z(n)

title = "Cray Data:"
a = acos(-1.0)
c = cmplx(a,a)
do i = 1, n
   x(i) = real(i)
   z(i) = cmplx(i,-i)
end do

open(unit=33, file="cray.dat", form="unformatted")
write(33) title
write(33) a
write(33) c
write(33) n
write(33) x(:)
write(33) z(:)
close(33)

end
```

5.2 Module cray_binary

The following Fortran module, cray_binary, is useful for reading real and complex numbers from a Cray file on an Origin.

```
module cray_binary

implicit none

private
public :: read_cray

interface read_cray
module procedure read_real, read_real_array,
$     read_complex, read_complex_array
end interface

integer, parameter :: real_type = 3
```

```
 integer, parameter :: complex_type = 4
 integer, parameter :: real_size = 64
 integer, parameter :: complex_size = 128

 contains


 subroutine read_real(unit, value)

 integer, intent(in) :: unit
 real(8), intent(out) :: value

 integer :: status, cry2mips
 integer(8) :: temp

 read(unit) temp
 status = cry2mips(real_type, 1, temp, 0, value, 1,
$     real_size, real_size)
 if (status .ne. 0) then
    print *, "CRY2MIPS FAILED WITH STATUS", status
    stop "IN READ_REAL (READ_CRAY)"
 end if

 end subroutine


 subroutine read_real_array(unit, value)

 integer, intent(in) :: unit
 real(8), intent(out) :: value(:)

 integer :: n, status, cry2mips
 integer(8), allocatable :: temp(:)

 n = size(value, 1)
 allocate(temp(n))
 read(unit) temp(:)
 status = cry2mips(real_type, n, temp(1), 0, value(1), 1,
$     real_size, real_size)
 if (status .ne. 0) then
    print *, "CRY2MIPS FAILED WITH STATUS", status
    stop "IN READ_REAL (READ_CRAY)"
 end if
 deallocate(temp)

 end subroutine


 subroutine read_complex(unit, value)

 integer, intent(in) :: unit
```

```
 complex(8), intent(out) :: value

 integer :: status, cry2mips
 integer(8) :: temp(2)

 read(unit) temp(:)
 status = cry2mips(complex_type, 1, temp(1), 0, value, 1,
$     complex_size, complex_size)
 if (status .ne. 0) then
    print *, "CRY2MIPS FAILED WITH STATUS", status
    stop "IN READ_COMPLEX (READ_CRAY)"
 end if

 end subroutine


 subroutine read_complex_array(unit, value)

 integer, intent(in) :: unit
 complex(8), intent(out) :: value(:)

 integer :: n, status, cry2mips
 integer(8), allocatable :: temp(:,:)

 n = size(value, 1)
 allocate(temp(2,n))
 read(unit) temp(:,:)
 status = cry2mips(complex_type, n, temp(1,1), 0, value(1), 1,
$     complex_size, complex_size)
 if (status .ne. 0) then
    print *, "CRY2MIPS FAILED WITH STATUS", status
    stop "IN READ_COMPLEX (READ_CRAY)"
 end if
 deallocate(temp)

 end subroutine


 end module
```

5.3 Program `read_cray_binary`

The following Fortran program, `read_cray_binary`, reads and prints data from "`cray.dat`". It uses the `cray_binary` module from Section 5.2 and requires the appropriate `assign` command before execution.

```
 program read_cray_binary

 use cray_binary

 implicit none
```

```
integer n, i
character(len=10) title
real(8) a
real(8), allocatable :: x(:)
complex(8) c
complex(8), allocatable :: z(:)

open(unit=33, file="cray.dat", form="unformatted")
read(33) title
print *, title
call read_cray(33, a)
print *, "A =", a
call read_cray(33, c)
print *, "C =", c
read(33) n
print *, "N =", n
allocate(x(n), z(n))
call read_cray(33, x)
print *, "X =", x
call read_cray(33, z)
print *, "Z =", z
close(33)

end program
```

References

All material in this document not derived from experiments by the author comes from information supplied in the `man` pages for `assign`, `f90`, `f77`, and `cry2mips`.